3D Color Model Reconstruction by Using the Kinect

Peng Liu

[liux1342@umn.edu](mailto:liux1342@umn.edu)

Department of Computer Science

Abstract

3D color models of the real world, such as indoor scenes, the human body and objects around our daily life, are useful for applications in computer games, animation and virtual reality. Accurate model recreation systems, however, either require very expensive scanners or take a lot of time by using modeling software. Although the cheap depth sensor, like the Microsoft Kinect, which has the potential to facilitate the model creation task, has attracted tremendous attention in academia, most research efforts were aimed at doing pose tracking or robotic navigation, and few have focused on using these depth sensors to do color model reconstruction. In this paper we present such a depth sensor based system, which uses a Kinect to do color model reconstruction. The system uses an incremental updating scheme based on the KinectFusion algorithm[1]. With this fast and affordable (only a Kinect and decent graphic card are needed) color modeling system, creating real world objects can be done more easily than before, which promises a more efficient development processing in model creation related applications.

1 Introduction

In many computer graphics related applications, like animation, video games or even research areas like virtual reality and augmented reality, the task of creating a realistic model is either performed by graphics specialists, who employ specialized design software, or by using 3D scanning technology to capture a real object. Either way, the work is expensive (both the advanced design software and the scanner are expensive) and time consuming.

Recently, with it’s invention, the Kinect, a cheap depth sensor, has drawn extensive attention in both industry and academia. Compared with conventional 3D scanners, which are capable of capturing depth and image data at video rate, the Kinect is more compact, cheaper and easier to use. Thus, some research groups have tried to use the Kinect as a 3D scanner. However, many of these applications (especially those that only use a few frames from the depth sensor) suffer from the comparably low resolution and depth accuracy of the depth sensor; therefore, numerous efforts have been put into increasing the quality of the depth image. For example, Sun [2] proposed an automatic face modeling system which uses the Kinect to do high resolution 3D facial modeling. In order to address the issue of low quality depth data, they built a resolution enhancement framework which uses image segmentation and smoothing techniques to capture a high resolution partial face model. Cui [3] described a method to improve the quality of the incoming depth map through a super resolution algorithm which takes several low resolution depth images of the target from the same view port and outputs one high resolution depth image. Weiss [4] presented a system to perform 3D body scans from a single Kinect, circumventing the low quality depth image problem by using a human body template. Their system takes several depth snapshots of a target human body, then, based on the silhouette of each snapshot, their system finds the body pose by fitting a template body onto the silhouette and updates each partial model of the human body according to the pose of each part of the template body.

The aforementioned systems only use a single Kinect to capture target models, and all have some limitations: Sun’s system can only generate partial facial models; Weiss’s system cannot work with subjects with clothing on; and Cui’s system takes a long time to process and requires a rotating disk which further limits its applications. To extend the usage of the Kinect, some researchers tried to use multiple Kinects working together to get better results. Tong [5] created a multi-Kinect based body scanner. Their system uses three fixed Kinects looking at different parts of the human body without overlapping area. The target needs to stand on a fixed turntable for a 360 degree scan before the system can output the complete model of the human body. A major limitation of their system is that it can only create models for objects which can be put onto the turntable. The system also requires a setup process that further limits its applicability. The above systems all have a limitation in that they have separated the capturing phase and processing phase, which is not preferable for real-time applications such as immersive virtual reality. To address this issue, Alexiadis [6] and his team developed a system which uses multiple Kinects to do real-time full 3D reconstruction of objects. Their system receives multiple depth data streams from multiple Kinects through different viewpoints, applies a coarse-to-fine registration algorithm to register different partial surfaces of the target, and merges separate meshes onto a single 3D surface. However, due to the high noise level of the live depth map from the Kinect, the quality of their generated mesh is hardly acceptable for immersive virtual reality applications.

A great leap in depth sensor based scan systems was made in 2011 when Newcombe [1] introduced the KinectFusion algorithm, which uses a dense mapping approach to register between consecutive depth maps, and then fuses the incoming depth surfaces into a truncated distance volume for model generation. That was the first real-time dense model fusing system with unprecedented accuracy. It is truly remarkable that with just one hand-held depth sensor, their system can produce a very accurate geometry model. However, their system cannot do a full color model reconstruction, and due to their implementation decisions, their system can only work on systems with an Nvidia Graphics card with CUDA feature installed. In this paper, we present a system, based on the KinectFusion algorithm, which uses both the depth data stream and the RGB data stream to create a color model of any object or indoor scene without relying on any CUDA implementation.

2 Approach

In this section we describe our system in detail. The framework of our system consists of several key processing steps: raw data acquisition, preprocessing, surface extraction, surface alignment, volume updating, and visualization — see Figure 1 for information about system work-flow. In the following subsection we will explain each key step.

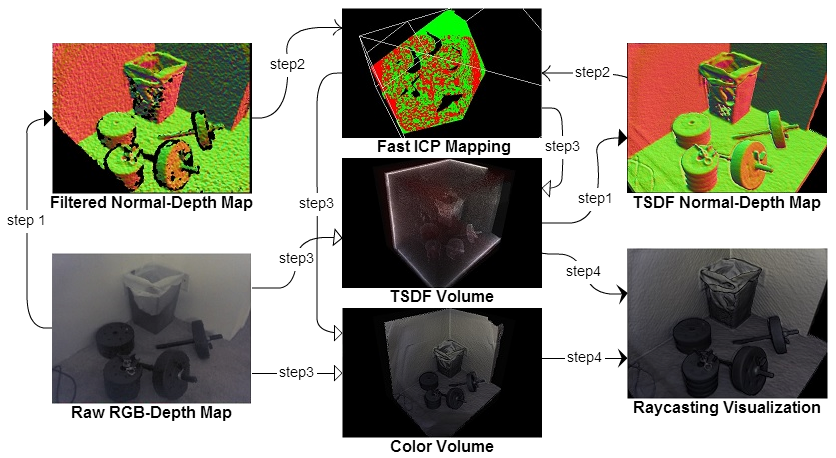


Figure 1: The overall system work-flow. After each new raw RGB-Depth map has been received, the system will perform an operation sequence to integrate new data into the implicit target model. In step 1, raw data will be filtered to get rid of noise, then a normal-depth map will be generated. At the same time, another normal-depth map will be extracted from the TSDF volume for mapping. In step 2, the fast ICP algorithm will find the rigid transform matrix between two normal-depth maps to get the pose of the Kinect. In step 3, with the transform matrix and the raw RGB-Depth data, both the TSDF and Color Volumes will be updated to fuse a new measured surface. Then, in step 4, a ray-casting algorithm will sample both the TSDF and Color Volumes to generate a Phong shaded image of the target model.

2.1 RGB-Depth acquisition

The Kinect has a structured light-based depth sensor along with a commodity RGB camera. The depth sensor is capable of generating a 640x480 depth map at 30 frames per second. At each frame, the Kinect will stream out both the depth map and the RGB image; in order to reconstruct the color 3D model, the depth map and the color image must be registered. However, the offset between the RGB sensor and the depth sensor makes it impossible to correctly register the depth and RGB map for all depth levels, so artifacts will appear around the edges when assigning colors to the model. Our system address this problem by simply not updating colors near edges. However, when we have small objects very near the sensors, suck as cables in front of the Kinect, the color artifacts can still appear. A more sophisticated method may be developed in the future to address this issue.

2.2 Pre-Processing

With the low cost of the Kinect, the quality of the depth sensor is acceptable, but there are still some challenges for the Kinect to do an accurate 3D scan. One major problem is the high noise level of the depth data. The Kinect was mainly designed to track human body movement from 1 to 4 meters away, so the high noise level would not affect the result too much. But for reconstructing a model with details, noisy depth data causes problems. Although we could average out the noise by using multiple scans for one surface measurement, the pose estimation phase (finding the rigid body transformation to register two depth maps) will be badly affected by these noises. To get an accurate pose estimation, our system applies a bilateral filter on each incoming depth map to remove the noise, while at the same time keeping the sharp edges of the depth map (the surface discontinuity must be kept to correctly reconstruct the geometry). The bilateral filter adopted in our system is the separable version, in which the 7x7 kernel pass was separated into two 1x7 (horizontal and vertical) kernel passes. To maintain real-time performance, we utilized the power of the GPU by performing the filter algorithm purely on the GPU in forms of pixel shader. The filtered depth map was only used for pose estimation (in Fast ICP process), while during the updating phase, raw RGB-Depth data was used instead in order to keep the high frequency detail.

2.3 Surface alignment

The correctness of incremental model updating is ensured by aligning the incoming partial surfaces with the model surface we already have. This problem has been well studied in robotics as SLAM(simultaneous localization and mapping), and the most popular solution to align 3D surfaces based on geometry information(with initial estimate pose known) is the ICP (iterative closest point) algorithm. After the introduction of the ICP algorithm, many variants have been developed. Rusinkiewicz [7] provided a well organized re- view and efficient comparison on some of the most popular variants of the ICP algorithm. In Rusinkiewicz’s review, there are basically six stages of the ICP algorithm: Selection, Match- ing, Weighting, Rejecting, Error metric assignment and Error metric minimizing. Most ICP variants differentiate from each other by using a different method in at least one stage of the ICP algorithm routine. According to their research and our system specification, we chose

4

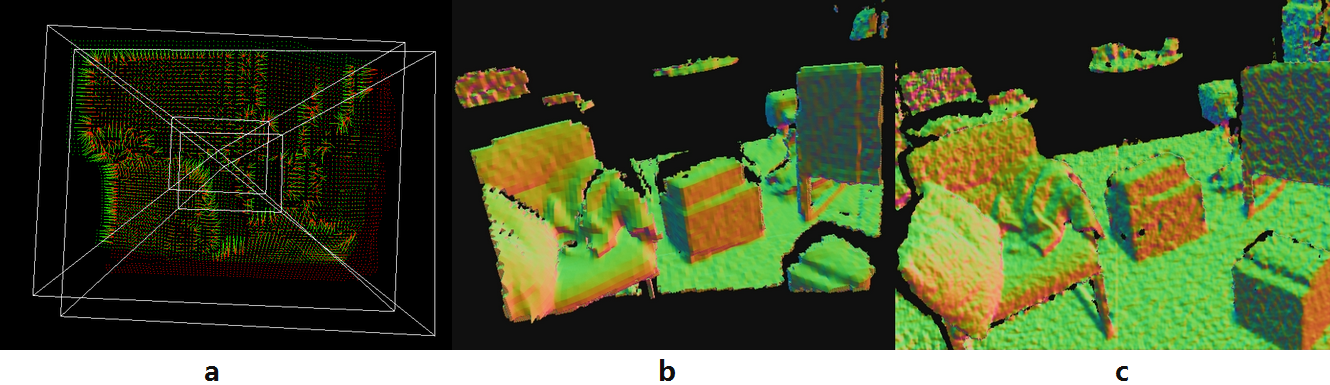


Figure 2: (a) demonstrates the closest corresponding matching phase, in which every green point will search in a neighboring area in the red point cloud to find the closest corresponding point, which is very time consuming even when implemented by the GPU. (b) shows the normal shaded model being reconstructed while (c) is the live depth map at that time. There is nearly a 2 second delay during the updating phase; thus the point-to-point version of our system cannot achieve real-time performance, but with more powerful hardware it may be possible.

two different ICP algorithms as candidates. Both use all available points during the selecting phases and use the same criteria for the rejecting and weighting phase; one uses modified nearest point for matching and point-to-point distance as an error metric, the other uses projecting rays for point matching and point-to-plane distance as an error metric. The reasoning behind our decision is that [7] suggests the point-to-point method has the lower execution time per iteration, while the point-to-plane method, although it has longer per iteration execution time, has better convergence rate, which means it will iterate for less time to get the result.

The point-to-point method we tried used a modified closest point matching strategy, in which each point will search for a closest point around it in the target depth map instead of the whole depth map. Our result shows that in order to allow 3 degrees of rotational movement between neighboring depth maps (it is quite common that we can rotate the Kinect by hand in 1*/*30 second), the search area in a typical 640x480 depth map has to be at least 50x50. Even a sample rate of just a 0*.*252 (for every 4x4 points, sample only 1 point) will require 100 iterations for each vertex, and our system can only get 7 FPS on such a setting. So actually the point-to-point version cannot achieve real-time performance. See Figure 2 for details.

2.3.1 Fast ICP alogrithm

The point-to-plane iterative correspondent point variant is actually called Fast ICP, intro- duced by Chen and Medioni [8]. The differences between their algorithm and the standard ICP are in how they do points matching and the error metric they use. As shown in Figure 3, the Fast ICP uses a projection ray from the viewpoint to find a matching point on the target surface while the standard version’s matching routine searches for the closest point on the whole target surface.

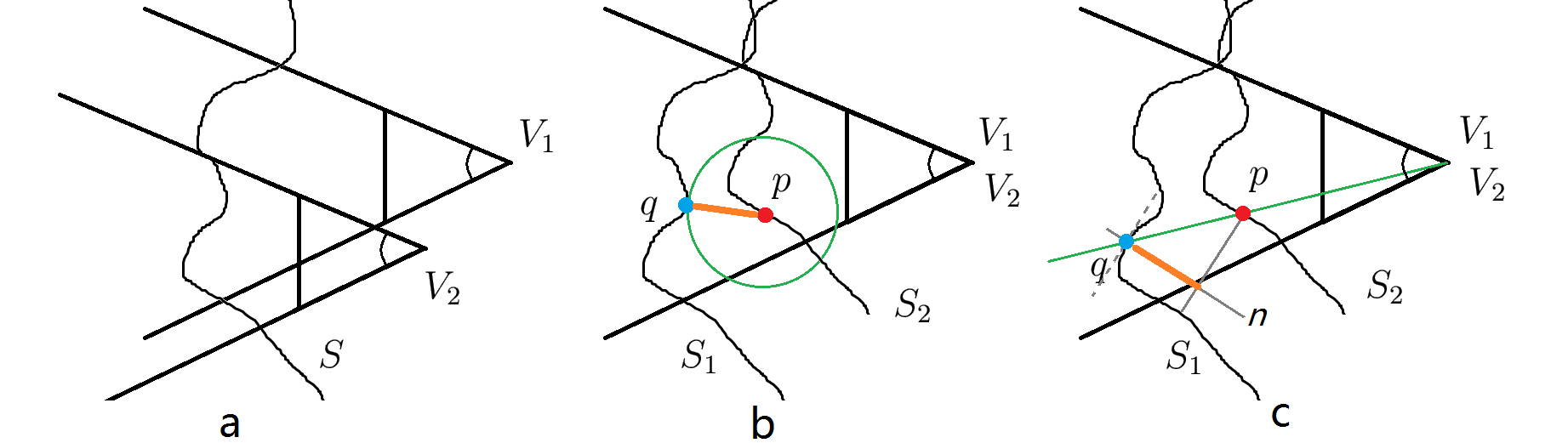


Figure 3: Demonstration of the differences in the matching stage of the standard ICP algorithm and the Fast ICP algorithm used in our system. In (a) we measure a surface *S* from different viewports *V*1 ,*V*2, and get the surface measurement *S*1,*S*2. In (b) we demonstrate the standard ICP matching strategy: for point *p* in *S*2, the corresponding point *q* is the closest point to *p*, and the error is the distance between *p* and *q*. In (c) the Fast ICP uses the projective data association algorithm[8] to obtain the correspondent point, and the error is the distance between *p* and the tangent plane of *q*

When having two depth maps in hand, the matching phase is quite simple: since the two depth maps have the same resolution, following the projective association algorithm, the corresponding point pair actually have the same texture index (we implement this algorithm in the form of a pixel shader, and we pass the two depth maps into the pixel shader as a texture object). So actually there is no computational cost in the matching phase.

However, the Fast ICP algorithm uses a point-to-plane error metric for the final minimizing stage(Equation 1) which is more complex than the standard point-to-point error metric version(Equation 2, *R* is the rotation matrix while *t* is the translation vector, and *n* is the normal vector at point *qi* ) and requires the knowledge of its normal map. See Figure 3 for details.

The normal map for the Fast ICP algorithm is generated by calculating the cross product of two edge vectors consisting of the current point and its two neighbor points. The result of our system shows that the increased workload of the Fast ICP algorithm is paid off by the reduced iterations and it actually achieves real-time performance.

2.3.2 Pose estimation

The Kinect’s initial 6DOF pose is defined as centered at (0*,* 0*,* 0) looking along the positive

*z* axis in global space, so the pose of a *k*th depth frame is defined by a transformation matrix

[*Rk* tk ]

*Tk* = 0 1

where *Rk* is a 3x3 rotation matrix and tk is the translation vector. This matrix maps the surface in Kinect’s local space onto the global space. Thus every measured point *p* in local space can be transformed into global space as pg = *Tk* pk .

The depth map is actually one channel image, and each texel (*u, v*) only stores the *z* axis distance from the nearest surface to the depth sensor *D*(*u, v*). In order to get the 3D coordinate of each point *V* (*p*) in the depth map, another back projection matrix is needed. Once we know the intrinsic camera calibration parameters, we can build the function

*V* (*D*(*u, v*)) = ( (*u − cu*)*D*(*u, v*) *,* (*v − cv* )*D*(*u, v*) *, D*(*u, v*))

*fu fv*

to map the surface measurement *D*(*u, v*) to the actual 3D point *V* (*D*(*u, v*)) in local space (*cu, cv* are the *u, v* indices of the center point in the depth texture, while *fu, fv* are the focal lengths).

The Kinect pose transformation matrix in frame *k* is the one that aligns the *k*th measured surface with the global reference model. So with two point clouds(surfaces) in hand, we can get the rigid-body transformation matrix by solving an optimization problem which is aimed at finding the optimal matrix *R* and vector *t* to minimize the alignment error. The error metric used in our system as mentioned above is the point-to-plane distance, so with a collection of points (*pi , qi* ) and normals *ni* , we need to minimize the following alignment error

*E* = ∑ [(*Rpi* + *t − qi* ) *· ni* ] (3)

*i*

where *R* is the rotation matrix and *t* is the translation vector. Since there are trigonometric functions in the rotation matrix, this optimization system is nonlinear. However, it is possible to linearize this system by utilizing the fact that the rotation movement will be small between two consecutive frames, since we are hand-holding the Kinect. So it is safe to approximate *cosθ* as 1 and *sinθ* as *θ*. Then our rotation matrix *R* can be approximated as

*R* = *Rx,α × Ry,β × Rz,γ*

for rotations *α*,*β* and *γ* around axis *x*,*y* and *z*.

Substituting (4) into (3), and rewriting some terms we obtain

*E* = ∑ [(*pi −qi* )*·ni* +*t·ni* +*α*(*pi,y ·ni,z −pi,z ·ni,y* )+*β*(*pi,z ·ni,x −pi,x ·ni,z* )+*γ*(*pi,x ·ni,y −pi,y ·ni,x* )]2

*i*

Defining *r* = [*α, β, γ*], and simplifying (5) we get the alignment error function as

*E* = ∑ [(*pi − qi* ) *· ni* + *t · ni* + *r ·* (*pi × ni* )]2 (5)

To minimize *E* with respect to r and t(specifically: *α, β, γ, tx, ty , tz* ), we set the partial derivatives to zero:

*∂E* = ∑ 2*ni,x* [(*pi − qi* ) *· ni* + *r ·* (*pi × ni* )] = 0

Defining *ci* = *pi × ni* and reorganizing these equations in matrix form we get

some formulas

To solve this linear system, we use the Cholesky decomposition(given the fact that the 6x6 matrix on the left is symmetric). Then the result *α, β, γ, tx, ty , tz* is the optimal incremental rotation and translation parameters. Thus real-time pose tracking was accomplished by repeatedly multiplying the current transformation matrix with the incremental matrix.

In our system, all per-point computation and summation is implemented as a pixel shader running on the GPU in order to get real-time performance, while the Cholesky decom- position is done by using the Eigen library on the host CPU. The target point cloud(*pi* ) is extracted from the TSDF(mentioned in the next section), while the model point cloud(*qi* ) is taken from the Kinect’s depth sensor after pre-processing(bilateral filter). The reason we don’t simply use the adjacent depth map (depth maps from frame *k* and frame *k −* 1) for pose estimation is that the alignment error will accumulate, which can cause our system to fail after several seconds.

2.4 Model representation and updating

Once every new depth map gets aligned, it can be used to update the mesh model. One straight-forward way is to use mesh-based reconstruction algorithms to add all the aligned points into the output model, but a post-trimming process is needed since in each frame there are nearly

640x480 points that need to be added into our system which can overload the system very quickly. However, the trimming process may also be time consuming when the model is complex, and for our pose estimation processing to work correctly, we have to ensure the small movement assumption, which means we need to maintain constant execution time for each frame. So the mesh-based method is not suitable for our purposes; instead, as suggested by [1] we use the truncated sign distance function(TSDF) to represent the model.

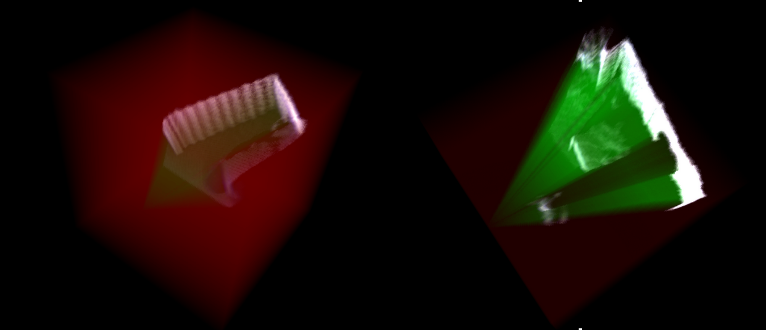


Figure 4: The TSDF volume is a 3D grid. The voxels in red are not visited by the Kinect’s frustum, while those in green are empty space, and the remianing voxels actually contain the model surface.

The SDF (signed distance function) was introduced by Curless [9], who used that function to fuse partial depth scans which avoided problems related to the mesh-based method as mentioned above. In SDF, the model was implicitly represented in a volume(a 3D grid), and each voxel in the SDF volume stores the distance to the nearest surface. The voxels which are outside the object have positive value while those inside have negative value. Thus the actual mesh is stored as the crossing zero surface in the volum (see Figure 4 for detail).

In our system, we use a truncated version of the SDF to store the model. For a voxel at *p* =

(*x, y, z*), the TSDF(*V* (*p*)) stores the truncated distance to the ”nearest” surface *Dtrunc* (*p*), and a weight *W eight*(*p*).

*V* (*p*) = [*D*(*p*)*, W* (*p*)]

To calculate the truncated distance for *V* (*p*), we first project voxel *p* onto the virtual Kinect’s image plane (which contains the depth map *D*), with the calculated pose matrix, and projec- tion matrix *Tk , Mp roj* to get the corresponding measured point *D*(*Mproj T −*1*p*), then truncate the distance between *V* (*p*) and *pdepth* by the predefined *T uncatedDist*. The truncated dis- tance is defined as

*Dtrunc* (*p*) =null

The weight *W eight*(*p*) is used to fuse incoming data with the previous one to get a smooth surface when updating the volume. The weighting strategy used in our system simply assigns each measurement with weight 1, and truncates the resulting weight to a predefined *M AX W EI GH T* to allow scene updating during model reconstruction.

The center of the TSDF volume was defined as the origin at *o*(0*,* 0*,* 0) in global space. With the proper voxel size and voxel resolution settings, each voxel in this volume can be easily located. When extracting the depth map from the TSDF for pose estimation, a virtual Kinect frustum will be created and transformed to the right place by using the previous transformation matrix (see equation 7). Our system will then ray cast the TSDF volume through the virtual Kinect’s frustum. The ray casting algorithm detects the crossing zero surface and generates a depth map for incoming surface alignment (see Figure 5). So in our system, the distance value in the TSDF is not the exact distance to the nearest surface, but a distance to the nearest surface along the projection ray. However, after several scans are integrated into the TSDF, the voxels near the surface have the approximately the same value as to the nearest surface. So during the surface extracting phase, we only use the voxels near the surface to calculate the crossing zero interface.

While updating the TSDF volume after the pose estimation process, the same ray casting algo- rithm will take place, with the exception that every voxel that intersects with the rays will be updated. For each voxel position *p* along the rays in frame *k*, the previous value on that voxel was extracted: *Dtrunc,k−*1(*p*)*, W eightk−*1 (*p*). Then, with the new measured distance *D*(*p*), we updated the volume by using the following formula:

*W eightk* (*p*) = *min*(*W eightk−*1 (*p*) + 1*, M AX W EI GH T* )

At the same time, the color volume is also updated. As mentioned in the RGB-Depth acquisition section, we don’t update those voxels which are near the silhouette of the incoming RGB-Depth map to avoid artifacts. Also, with the previous value in color volume *C olork−*1 (*p*) the color updating process uses weigh sto avoid turbulence in color representation:

*C olork* (*p*) =*W eightk−*1(*p*)*C olork−*1 (*p*) + *C olor*(*p*)

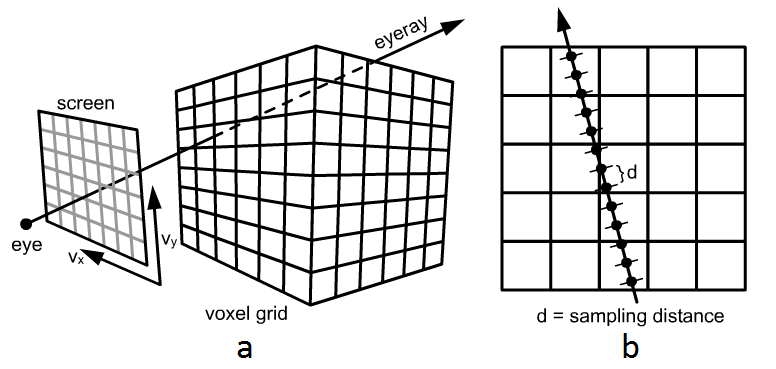


Figure 5: (a) With a known frustum pose, a projection screen(screen) and viewpoint(eye) can be uniquely located, then for each pixel in the projection screen, an eyeray will emit from the eye through that pixel into the voxel grid. (b) Each eyeray will sample the vol- ume along its way, and the surface interface is detected by finding the two neighboring sam- ples which have negative production(one is greater than 0 while the other is less than

0), then a linear interpolation will find the exact crossing zero surface. (image comes [from:http://johnrichie.com/V2/richie/isosurface/volume.](http://johnrichie.com/V2/richie/isosurface/volume.html)html)

As described above, a lot of voxels will be updated (typically, in our system both the TSDF and Color volume have a size of 3823 voxels, and during each frame nearly 70% of them will be updated); to maintain interactive FPS, all the extracting and updating phases were implemented as a pixel shader program in our system to utilize the power of the massive parallelism of the GPU. The result shows that our system can maintain a 25 FPS performance even with a 5123 volume size for both the TSDF and Color volume (that is 40ms per frame execution time including rendering the model from volume), which is pretty good given the fact that the depth stream of the Kinect sensor is working at 30 FPS.

2.5 Volume visualization

Once the volume gets updated, visualization is very straightforward: we define a free virtual camera in our system, and using the extracting method described above to get the rendered image from the TSDF, the difference is that instead of passing in the tracked Kinect pose as input, we are passing in the free camera’s pose instead. Also, after finding the crossing zero point in the TSDF, we sample the color volume at the same place to get the color information for that point, then a Phong shading algorithm was used to shade the model. See Figure 6 for result.



Figure 6: color model under Phong shading

3 Results

In the experiment, our system was running on an Alienware m18x laptop with an Nvidia 680M graphics card attached with 2GB video memory. The volume resolution is set to 3823 with different voxel size. The result shows that, for most of the time, our system can maintain a 20 FPS rate even with 6 sub-texture rendering(as shown in Figure 7). There are two major problems in our system: the first problem is the drifting effect(when the Kinect is faced by only planar scene, our system is one rotation (around the normal of the planar surface) and two translations (on the planar surface) unstable); the other problem is the color issue (the specular highlight area moves when we changing the viewport, which made the resulting model brighter than it really was). But overall, our system is robust to most scene settings and camera motions. Figure 7 is one snapshot taken from our system when running.

4 Conclusions

In this paper, we presented a new way to generate color models for real world objects by using the Kinect as a scanner, which may have lots of applications in Computer Games, Animation, Virtual environment and Augmented reality like personal avatar generation, character designing, etc. Also, the real-time performance feature of our system has the potential to be used as a tracking system in both virtual reality or robotics applications. But there are still problems that need to be solved: as mentioned in the results section, the drifting effect may be addressed by using an RGB feature based registration method in the future so the system may choose which registration method to use based on the scene it faced. A more challening problem is the specular highlight problem, and if we go a little bit further, we may want to find a way to extract material information from both the RGB and depth sensor.

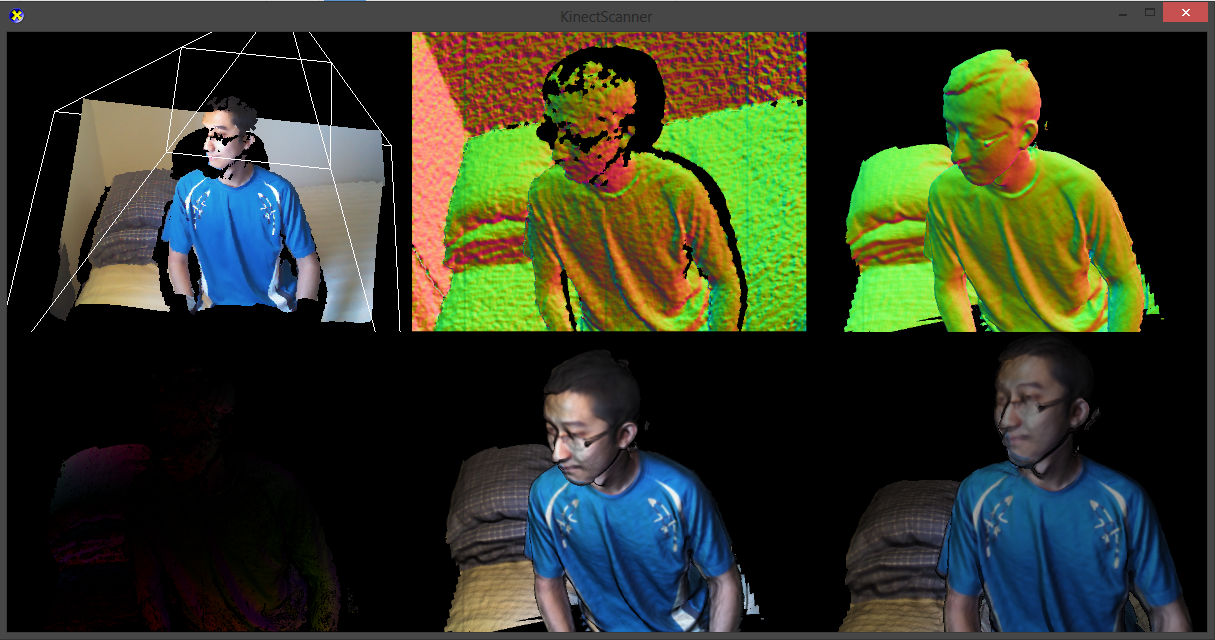


Figure 7: The sub images from left to right, top to bottom is: live RGB-D visualization with the Kinect live pose; live Normal-Depth map from the Kinect; Normal-Depth map extracted from TSDF; Error metric image; Phong shaded image from Kinect’s view; phone shaded image from free camera’s view

References

[1] R. a. Newcombe, A. J. Davison, S. Izadi, P. Kohli, O. Hilliges, J. Shotton, D. Molyneaux, S. Hodges, D. Kim, and A. Fitzgibbon, “KinectFusion: Real-time dense surface mapping and tracking,” *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pp. 127–136, Oct. 2011. [Online]. Available: [http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?ar](http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm)number=6162880

[2] Q. Sun, Y. Tang, P. Hu, and J. Peng, “Kinect-based automatic

3D high-resolution face modeling,” *2012 International Conference on Im- age Analysis and Signal Processing*, pp. 1–4, Nov. 2012. [Online]. Available: [http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?ar](http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm)number=6425065

[3] Y. Cui, S. Schuon, S. Thrun, D. Stricker, and C. Theobalt, “Algorithms for 3D shape scanning with a depth camera.” *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 5, pp. 1039–50, May 2013. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/23520250>

[4] A. Weiss, D. Hirshberg, and M. J. Black, “Consumer Depth Cameras for Computer

Vision,” 2013. [Online]. Available: [http://www.springerlink.com/index/10.1007/978-1-](http://www.springerlink.com/index/10.1007/)

4471-4640-7

[5] J. Tong, J. Zhou, L. Liu, Z. Pan, and H. Yan, “Scanning 3D full human bodies using

Kinects.” *IEEE transactions on visualization and computer graphics*, vol. 18, no. 4, pp.

643–50, Apr. 2012. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/22402692>

[6] D. S. Alexiadis, D. Zarpalas, and P. Daras, “Real-Time , Full 3-D Reconstruction of

Consumer Depth Cameras,” *IEEE Transactions on Multimedia*, vol. 15, no. 2, pp. 339–

358, 2013.

[7] S. Rusinkiewicz and M. Levoy, “Efficient variants of the ICP al- gorithm,” *Proceedings Third International Conference on 3-D Dig- ital Imaging and Modeling*, pp. 145–152. [Online]. Available: [http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?ar](http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm)number=924423

[8] Y. Chen and G. Medioni, “Object modeling by registration of multiple range images,” pp. 2724–2729, 1991. [Online]. Available: [http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?ar](http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm)number=132043

[9] B. Curless and M. Levoy, “A volumetric method for building complex models from range images,” *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH ’96*, pp. 303–312, 1996. [Online]. Available: [http://portal.acm.org/citation.cfm?doid=237170.237269](http://portal.acm.org/citation.cfm)